WILL HARWOOD

6 States and actions

Interactive devices and systems range from implants to wearable computers and from clocks in our central heating systems to flight management systems in aircraft. Behind all interaction programming is a simple, core idea struggling to get out. Behind all unnecessarily complex interaction programming is a simpler idea struggling to be heard.

This chapter introduces and explores the key concepts behind interaction programming, which hinge on finite state machines. It turns out that finite state machines are easy to draw, and drawing them gives us more insight into design—finite state machines are pretty good at showing what users do too, that is, they can be used for more than design—they can be used for *understanding*.

6.1 Key concepts

There are four key concepts in interaction: actions, states, indicators, and modes. Put very briefly: the user performs actions, which change the device state, which in turn control indicators. But users may not know exactly which state a system is in; they only know (if at all) which mode it is in.

What are actions? Users can do anything, and one could define actions in many different ways. For our purposes, we are specifically concerned with discrete, atomic actions. *Discrete* means you either have an action or you don't; there are no half-hearted actions, no double-effect actions, no long and short actions. If you want to have a button half-pressed or lightly touched (as on the focus lock on a camera), then you have to think of these as separate actions. In contrast, *atomic* means that there is nothing simpler to consider in a user's actions. User's actions may be quite sophisticated and different people, perhaps people with microscopes, might be able to unravel smaller things inside actions, but we aren't interested in those. If we decide that "changing a CD" is the action, then we are not interested (or we pretend we're not interested) in the fact that this action could be broken down into smaller steps: press the Open button, remove old CD, place new CD, press drawer shut. A neurologist might notice even smaller steps going on inside the user's fingers, nerve impulses, even things inside their head, and whatnot; real as they may be, these are not actions that concern us.

Of course, there is a wide range of discretion. We might be interested in brain computer interaction, then rather complex signals from the brain are the actions that concern us. We might want implants in our arms to respond to some other muscle twitching—perhaps to gain more hand movement—then we would consider twitching actions we previously overlooked.

Having said all that, it is easy to overlook important actions. Removing a battery from a mobile phone is an action, very similar to switching the device off—but not quite. On mine, if I receive a phone call but miss it for some reason, the number calling me is stored in a list of missed calls. When the phone is ringing, if I press the [Cancel] button the call is terminated but the number is not stored in the calls list. As a work-around, if the phone rings but I don't want to answer it, the best thing seems to be to remove the battery, as this stops it ringing *and* stores the number as a missed call, so I can ring back later. If we notice, then, that removing battery is an action, later when the device design is analyzed we should notice that it does something useful as a side-effect, which ought to be allocated to an explicit action, perhaps a button.

However, once we have decided on our set of actions, they are atomic and we do not explore within them.

We usually give actions names, and as a typographical convention in this book, I usually write actions as if they were buttons on a device. So I write [On] to indicate the *on* action. This looks like a button you can press, and although it often is, it needn't be. It could be a puff switch you blow into; it could even be the word "on" spoken into a microphone. There might be no name at all; for instance in the United States, flicking a switch up means [On], even though in the United Kingdom that action means [Off]!

Now, what do actions do? Actions change the state of the system, although sometimes the "change" might be to leave the system in the same state.

Unlike actions, which you could imagine watching happen, states are very abstract concepts. They do not exist, or at least there need be nothing identifiable anywhere that you could call a state. Instead, states make their presence felt through indicators.

Indicators might be lights that can be on or off, words, or sounds. Indicators show the user something has happened. Like actions, we consider indicators to be discrete and atomic. Except informally, the volume of a sound is not *an* indicator; level 0, level 1, level 2, or level 3 are different (closely related) indicators. A user may not be able to tell the difference between some indicators; level 3 sound might seem like level 0 sound in a quiet moment.

Indicators can be complex things if we wish. If you press the [Play] button on a DVD, you watch a video. The *entire* hour-long video is an indicator that pressing [Play] had the desired effect. As with actions, we may wish to split indicators up into constituent parts to understand them better. For example, you might prefer to have indicators for each of the scenes in a DVD video, rather than having a single play indicator—indeed some DVD players can display the scene number, and if so, these numbers are indicators.

In very simple devices, there is a straightforward relation between actions, states, and indicators. A light switch controlling a light is an example. The actions are to switch on or switch off; the states are on and off; and the *two* indicators are the light bulb being on or off. Unfortunately life is rarely so simple. In reality, the light bulb may be a dud and not work. Now the device always indicates off!

A mode is a set of states that behave in the same way. It is sometimes useful to distinguish between different sorts of modes, such as action modes or indicator modes, but usually the context makes it quite clear. For example, the mode "dud light bulb" is a set of states (two, at the present count) that happen to behave in a simple way: namely, whatever action the user does in this mode results in the indicator being off. We could consider this case an indicator mode: whatever happens, the indicator is off. Or we could consider it an action mode: whatever we do, actions do, as it happens, nothing.

States are abstract things. It is usual to give them names, or when we can't conveniently think of names, we give them numbers: 0, 1, 2, and so on—programmers like to count from zero. If we understand a device and we know what state it is in (17, for example), then we know what *any* sequence of actions will do to that device henceforth.

Of course, the device got itself into state 17 by some earlier sequence of actions. In an important sense, all of the sequences of actions that result, or could have resulted, in the device being in state 17 are equivalent. In fact, this is how we can define states mathematically: states are the equivalence classes of behavior.

▷ Graphs are the mathematical way to model states and actions. Graph theory will be covered in chapter 8, "Graphs." Simply put, states are vertices in a graph, and the arcs of the graph are the actions. Arcs go from a vertex to a vertex, just as an action goes from a state to a state.

6.1.1 States versus modes

While they are central concepts, modes and states often get confused—not least because people do not agree on exactly how the words should be defined, and they may not even notice they are using the words in different ways. I defined modes as certain sets of states; obviously if all modes are sets of single states, then modes and states may as well be the same concept—and indeed for some device designs modes and states *will* be the same.

It is worthwhile to take a moment to explore the distinctions in more detail.

Mode In a given mode, a *specific* action has a consistent effect (including, possibly, nothing). For example, if the Off button always switches a gadget off, then there is a mode "on" where the button behaves that way; the mode "off" probably behaves differently, certainly the Off button won't be able to turn the device off if it is already off! There are two modes in this case.

A common design slogan is, "Don't mode me out!"—meaning avoid modes that stop many actions having effects. Computers mode out users frequently,

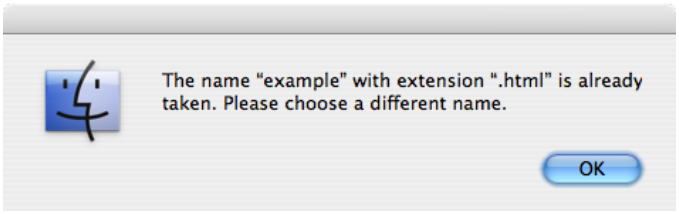


Figure 6.1: A familiar example of being moded out by a dialog box, here on an Apple Macintosh computer. To continue, the user *has* to choose OK. The program displaying this dialog box will allow the user to do nothing else; the user has been moded out. Here the user tried to change the name of a file, but they chose a name that was already taken. The dialog box tells them so, but the user is unable to look at file names to find out what the problem is; nor can the user try any other solutions, such as deleting or renaming the other file.

typically offering them a simple choice (the clearest example is the intrusive dialog box that says something . . . OK? and that's the *only* choice, to click on OK). When this happens, all other actions are inaccessible! Figure 6.1 (p. 166) gives an example. Fortunately this is becoming less of a problem on computers, but it is still a real issue for interactive devices, like DVD players and ticket machines.

Moding out is not always a problem. On a gun, to give a stark example, you might want a safety catch, which modes you out of the possibilities of unsafe or accidental use.

State In a given state, every action *always* works in *exactly* the same way (though perhaps doing nothing). In contrast, modes need not be very specific. For example, the "on mode" of a device only tells you what the Off key does—it doesn't tell you exactly what anything else will do, which knowing the state does.

Thus a mode tells you what one or perhaps *some* user actions will do, whereas a state tells you *everything* and *exactly* what the device will do. A state tells you what any button will do.

For example, all televisions have two very important modes: on and off. A TV can be in the on mode or in the off mode, and the TV modes broadly define what user actions will do. When a television is off, the only button that will work is generally the On/off button, which should switch it back on. If the TV is off, most buttons will do nothing, but when it is off, it can be switched on. When a TV is on, the On/off button will switch it off, but that's just about all we know about it. To know what all the other buttons do when the TV is on, you need to know more—you need to know more about the states or other modes within the on mode. The on mode only tells us what On/off will do; it doesn't tell us what any other buttons will do.

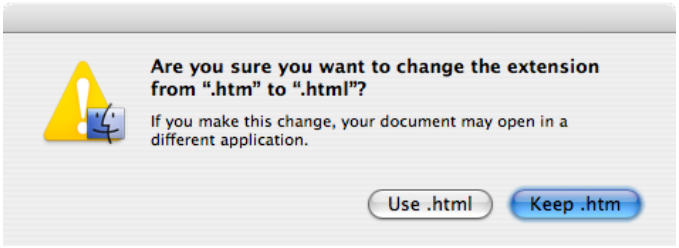

Are you sure you want to change the extension from ".htm" to ".html"?
If you make this change, your document may open in a different application.
Use .html
Keep .htm

Figure 6.2: As well as being modey, the dialog box in figure 6.1 is rude—rude in the sense of section 6.3.4 (p. 185). This dialog box shown here precedes it, giving you a choice that the computer will *immediately* forbid: if you choose "Use .html" it will tell you that you must choose another name, using the dialog box of figure 6.1. This dialog box is giving you a choice that it won't let you take.

If modes describe so much, how do states help?

Suppose you switch the television off. In this mode, all buttons probably do nothing apart from the On/off button. Yet when you switch the television on, it will (probably) go back to whatever channel you were watching when you switched it off; it might also know what the sound and brightness settings were before you switched off. Somehow the off mode "knows" what channel—and what sound level, what color and contrast settings, and so on—you had the television in before you switched it off. The off mode must remember these details somehow, and it does so by keeping track of the state before the TV was switched off. Yet whatever stuff the TV was keeping track of did not affect what On/off does; that's all the same mode.

If modes contain states, how many different individual states are there when everything is unraveled? *A lot*. There are all the off states, and all the on states. In *all* of those states the television has to keep track of its channel, so (for a simple 9-channel TV) there are 9 off states and 9 on states. That's 18 sorts of state already. Suppose there are 64 sound levels. Each of those 18 sorts of states has 64 individual variants to keep track of sound level. There's off in channel 1, and sound level 0; there's off in channel 1, and sound level 1; there's off in channel 1, and sound level 2 ... so far, there are $18 \times 64 = 1,152$ states *just for these features*. We've got over a thousand states before we've even got a fully featured television!

Clearly when we design a device, we don't want to think about *all* the states all the time as there are just too many of them. Instead, we tend to think about modes as we concentrate on individual actions, button presses or whatever, that the user can do.

Very often, using a gadget is so straightforward that it is hard to keep the state and mode concepts clearly in focus as two different things. In particular, humans are very good at chunking ideas together to make more powerful ideas, and this happens with actions, modes, and states. If a pair of actions, say, are done often enough, the user behaves as if they were a single action, chunked together. Often

Box 6.1 Syringe pumps A syringe pump is a device designed to operate a medical syringe automatically, to give a patient a controlled dose of a drug over a period of time. They are often used in pain control, where regular doses of painkiller is needed. Some pumps are portable and can be used and carried around by a patient.

In one case, a pump was set to deliver 0.1ml/hr (a tenth of a milliliter of drug per hour) to a baby. However, the syringe pump had not been activated to start delivery of the drug. The nurse had told the pump to deliver 0.1ml/hr and then completed her task. If you (pretending to be a pump) had been told to deliver 0.1ml/hr, you'd be expected to do it. The pump didn't, and wanted further confirmation from the user—causing a post-completion error. After two minutes, the pump starts beeping, to warn the user that it has been told to do something that hasn't been confirmed. To stop the noise, the nurse reached over and pressed a button to cancel the beeping. Unfortunately, the baby died shortly afterward.

The nurse had pressed not [Cancel] but [10], which canceled the beeping, but unfortunately also started the pump delivering at 10.1ml/hr—a hundred times faster. This was clearly a design error, particularly as post-completion errors are well understood.

▷ Post-completion errors are discussed in section 11.4 (p. 387). Another example is discussed in box 5.7, "The principle of least effort" (p. 147).

Other errors with syringe pumps have included the following, all avoidable by better interaction programming:

- 7ml/hr entered as 77ml/hr, by pressing [7] twice by mistake.
- 0.5ml/hr entered as 5ml/hr. The nurse here didn't press [·] hard enough, and didn't realize the mistake.
- 10ml/hr entered as 200ml/hr. Here the cause was a partly-obscured display. Displays should be legible, and sound feedback should be provided—using a synthesized voice would be easy.
- A pump defaults to 0.1mg/ml, but the syringe was filled with a concentration of 1mg/ml, so the patient got a dose ten times too high.

since no interesting effect happens until several actions have happened, the entire sequence of actions seems a single event so far as the user is concerned. Whether this matters or not is a question for the user, designer, and application. In the long run, a user might be confused and then make innocent mistakes. Chunking is also an issue for us, as interaction designers. Do we want to treat every state as different, or do we want to gloss some differences? We might treat playing all the tracks on a CD as different states, but do we also want to treat all the positions within the tracks as different? Are we even worried about different tracks? On the one hand, we don't want to be overwhelmed by detail, on the other hand, we want to be precise and design good systems.

6.1.2 Examples of state machines

If we make the further proviso, along with being atomic and discrete, that the devices we are concerned with are finite, then we have defined ourselves a partic-

ularly useful form of finite state machine, or FSM. Finite state machines turn out to be very useful.

Many things can be described using states and actions.

- A web site has HTML pages (states) and links to other pages. Links are the actions to go to other pages.
- A flashlight has states, such as ON and OFF. It has actions, such as SWITCH ON and SWITCH OFF.
- An aircraft landing system has numerous states, including ON GROUND and IN AIR. Its actions will include LAND and TAKE OFF.
- A television set has states such as OFF, WATCHING CHANNEL 1, WATCHING CHANNEL 5, and it has actions such as ON/OFF (a combined switch on and switch off) and NEXT CHANNEL.
- A dog has states such as sitting, walking to heal, lying down. It has actions, usually voice commands, "sit!" "heal!" and so on. Many dogs have modes, such as "obedient," and "deaf."
- A computer has lots of states, and the actions on it include mouse clicking as well as switching it on and off.
- Many systems, from mobile phones and televisions to car radios, and desktop computers have menus. Each menu item is a state: the state where that menu item is displayed (or said) to the user; each action is either selecting another menu item, or activating the current selection.

Three points follow from this list.

First, the number of states and actions we consider depends on what we (as interaction programmers) are trying to achieve. For example, a basic understanding of a flashlight is possible with just two states (on and off), but we get a more detailed understanding by considering more states. We need enough states to be realistic, but there are, in the limit, an arbitrary number of states: thus, BROKEN might represent countless specific ways in which a flashlight can be broken. In principle, these sorts of state cannot be counted: there is no end to them and no systematic way of deciding what they are.

Second, some states are irrelevant to user needs (users don't much care what states are—they are details inside the device). For example, a flashlight behaves much the same whether it is on a table or in a car, whether it is in the dark or in the sunshine. For most purposes, there is no need to distinguish these different physical states. Many times, we do not bother about sometimes relevant possibilities, like "battery nearly dead," which is an intermediate state between battery OK and battery dead.

Finally, a computer, at least when considered as a whole, has far too many states to deal with, even though in principle they can be counted off. Instead, we take large clumps of states and treat them together—just as a user would ignore all the

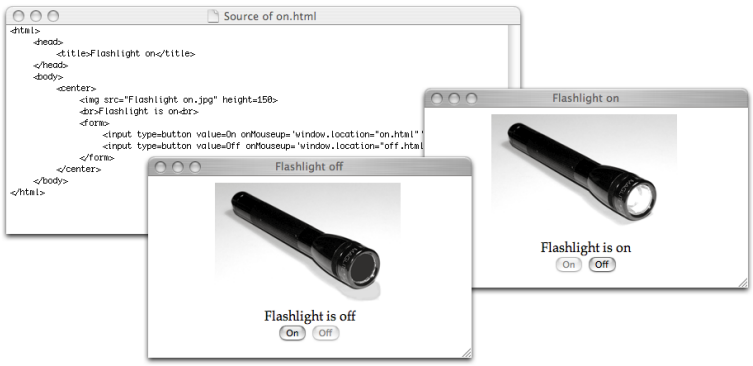


Figure 6.3: Interactive devices and web sites are both finite state machines and are thus essentially the same thing. Here, a trivial two-page web site behaves exactly like a simple flashlight. The web site has a page for the flashlight being on, and a page for the flashlight being off. The two pages are linked together by the form buttons, which are behaving like the flashlight's switch.

activities the computer is up to that have nothing directly to do with the present task.

A gadget can be understood using states and actions, and so can a web site. Can we get any mileage out of this similarity? Yes, we can. You could build a web site that behaved exactly like a gadget: when you click on the web site (using a browser) you would be taken to new pages that show you what would have happened if you'd pressed the corresponding button on the gadget. The web site might be just descriptive text, but it could be made up from lots of pictures (perhaps image maps), and if you did it really well, the web site would look and behave *exactly* like the device it described.

▷ We will build web sites from devices in chapter 9, "A framework for design."

You could imagine having a web site that *looked* like the gadget and a corresponding web site that told you what the gadget was supposed to do in each state. In fact there could be the obvious links between the two sites, and from any picture of the user interface you could immediately get the descriptive help for that state. For any bit of the help text, you could immediately transfer over to the simulation of the gadget, and it would be in the right state.

Of course it's quite irrelevant really that these things are web sites. We could have a *real* gadget that does exactly what it is supposed to do, and it could have a help button on it that displayed the "web help" on the gadget's own screen. If it was something like a DVD player or video recorder, it could use the TV screen that's already there.

This easy way to slide between different representations is the power of finite state machines. It doesn't matter whether they are DVD players, TVs, web sites, computers, or bits of string.

There remain three more interesting points:

- Because finite state machines are finite, they turn out to be easy to analyze thoroughly.
- Finite state machines cover an enormous range of interactive devices. The design, implementation, and usability of such devices can be supported and analyzed very well by finite state machine models.
- Many people dismiss finite state machines as being rather too simple-minded.

The first two points are advantages; the third is a misunderstanding that needs correcting. Most people who already know about finite state machines came across them in text books on theoretical computer science. Finite state machines are discussed (along with regular expressions) and then you turn over the page to the next chapter on Turing Machines. Turing Machines are infinite machines, and they are a good theoretical model of computers. In comparison, finite state machines seem pretty tame. You can prove finite state machines cannot do all sorts of things that Turing Machines find easy. Therefore, one might dismiss finite state machines as merely a conceptual stepping stone to interesting things.

This is a misconception. Finite state machines can have any number of states, and they can have more states than any real computer can; they are as powerful as any computer you can build. Turing Machines are, in fact, unrealistically powerful—they correspond to nothing that can be built in a physical universe. However, from a mathematical point of view, when you have huge numbers of states it becomes easier to think and prove things if you take the number of states to be infinite—you don't need to worry about the exact number of states. For example, if you prove a Turing Machine cannot do something then you can be certain no computer could do it either. On the other hand, if you proved a finite state machine couldn't do something, could one with more states do it?

If we allow our finite state machines to have billions and billions of states, in a purist sense we would be cheating. With *that* many states, it would be far easier not to count and to pretend that we could have any number of states—we'd be treating them as infinite machines for all practical purposes. If we are not going to cheat, then, we have to make some compromises so that our very definite idea of finiteness does not run away.

This book is about interactive systems, and we need finite state machines to talk about user interfaces, to describe what the user thinks the devices are doing—in a way that also makes sense to interaction programmers. Viewed like this, a very good argument for finite state machines (for our purposes) is that users don't understand bigger or more general things—and even fifty or a hundred states are problematic. There are plenty of devices with ten to twenty states that are very difficult to use. Turing Machines and other sophisticated models of computing don't describe what users think about.

▷ Box 12.3, "Defending finite state machines" (p. 430) reviews more defenses for finite state machines; the main one is that they encourage simplicity—which is the topic of the next section here.

Turing Machines are not the only possibility. There are very many alternatives to finite state machines and although they are important topics in their own right, this book doesn't cover pushdown automata (PDAs), Petri nets, and a variety of other viable alternatives, such as using programming languages (including modeling languages like CSP, Spin, SMV, and even regular expressions) that compile to finite state machines.

6.1.3 Ensuring simplicity

Discrete means you either have it or you don't—in contrast to *discreet*, which means unobtrusive. We have been talking finite discrete systems, indeed, all our examples have had a relatively few number of states. Discrete mathematics is the mathematics of things that you either have or don't have: an orange is either in a set of fruit or it isn't, so sets are part of discrete maths. In contrast, the weight of an orange is a real number, like 0.23, and that wouldn't be very interesting to a discrete mathematician.

Our discrete approach therefore in principle excludes continuous systems, such as virtual reality, gestures, speech, and even time delays from consideration. Such systems are not accurately understood by being in either one state or another, because their "states" merge with one another. Rather than using whole numbers, we'd more realistically describe these systems using real numbers.

It might seem that this book excludes a rather large (and interesting) area from our coverage of interaction programming! Philosophically, there are always going to be things you can't do with any approach, so we needn't worry *in principle* too much. More pragmatically, though, we can do a great deal with discrete systems—and fortunately when we need to, we can approximate continuous systems as accurately as we like by remaining discrete. Certainly as designers we can approximate continuous systems better than users can, which is perhaps all that matters.

Many gadgets, like video or DVD recorders, are both discrete and continuous. As well as their buttons, which are either pressed or not pressed, they have rotating dials and complex timing features needed, for example to set program schedules. The position of a CD or a tape is essentially a continuous variable. If we are going to design devices like this, we *have* to handle continuity somehow. If we are going to ignore it, we have to be certain that we are not glossing over important design issues.

We have four main choices:

Limit We can treat continuous systems as the limit of discrete systems. Rather than having to deal with a continuous system, we can treat it as if it has a million states, say. The user won't be able to tell the difference. Actually, if we wrote a computer program to handle a continuous variable (such as the sound level), it would still have discrete levels, though probably under anybody's threshold to notice.

Mode We can treat the continuous bits of a system as a mode that we won't explore in detail. We can just consider the mode to somehow include

continuous states without going into detail. For example, we can consider the volume on a TV as either off or on, even though in the on mode there might really be a continuum of volume levels. For the purposes of design (and use), all those levels should work the same way—they are just one mode. In fact, it would be a pretty peculiar design that had a continuous feature that somehow behaved in a wide variety of ways and therefore needed lots of modes to describe it.

Forbid If a device requires more than a dozen or so states, then it is going to be a complicated device; if it needs thousands of states, then it is a very complicated device. As designers, we can *decide* that we are not going to countenance building overly complex devices. It is always best to begin with a device that is a small simple discrete system, *then* expand it with features if necessary.

Strategize . . .

▷ The fourth choice is to think at a strategic level—see part III and particularly chapter 12, "Grand design."

One of the main continuous issues we cannot avoid handling is time. On many devices, how long a user presses a button or how long a user waits before doing something is a key part of the design. How then can we handle time?

One approach is to say "time is bad" and refuse to design with any timing constraints. In fact, this is such a popular approach that systems like this have a special name: a system that essentially takes no time at all to respond to actions applied to it is called a reactive system, because it can be thought of providing instant reactions to the user.

Generally, pure reactive systems are easy to use because the user doesn't have to understand or worry about any timing issues. If a button is pressed, it does one thing. It doesn't matter how long or briefly the user presses. If you leave a reactive device alone, it won't forget what it was doing; the user can have a break and return to exactly where they were.

Reactive systems are much easier to use because things mean the same however slow (or distracted) or however fast the user is.

The problem is that reactive systems are an idealization; many common systems—like the mobile phones—*do* have delays and timing issues, and we must know how to proceed. People don't answer their phones immediately, so there is a delay between calling somebody and their answering (if they do). Or, when you switch a mobile phone on, it takes a moment to get working fully. If a user asks a DVD player to rewind its tape it can't happen *instantly*. In particular, with a DVD player if a user does [Rewind] [Stop] they can expect something quite different to happen than if they do [Rewind] *wait* [Stop]. In short, most things are not perfect reactive systems.

▷ Timeouts are discussed in box 6.2, "Timeouts" (p. 174).

We can make time itself into an action. A clock can tick, and ticks can be actions—the user doesn't have to do all actions on a device. The device can be acted on by its environment, and time is one of the things that can act independently. If we wanted to be imaginative, we could have a button on a device called

Box 6.2 Timeouts Many systems use timeouts: when the user does nothing for a second or two, or takes several seconds pressing a single button, different things happen.

The main case where timeouts are useful—but often misused—occurs in walk-up-and-use devices that are intended to be used by a queue of people. The second person does not want to start with the mess left by their predecessor, so a timeout is used to reset the system for the next person. Actually this would be far better done by a heat or other sort of body sensor than a time out: what happens if the first person is too slow or the second too fast?

When trivial things like airport toilets use proximity detectors to flush themselves when you move away from them, why can't more sophisticated gadgets? Why can't ticket machines detect people who are present and using them, or when they walk away and stop using them? Why guess with timeouts when you can do better?

My favorite example is my old JVC video recorder, which had a timeout that would reset the device when you were setting up UHF channels. In the couple of minutes you took to follow the manual, which you need to do something complex like setting up UHF channels, the video recorder would have timed out—guaranteeing you couldn't follow the manual's instructions!

▷ Section 3.8.3 (p. 75) described timeouts for a washing machine, and box 6.4, "Bad user interfaces earn money" (p. 191) shows the problem with burglar alarms. Ironically, the burglar alarm problem arises with a device that *does* have a sensor that "knows" when the user is around!

Tick and we'd employ a little gnome to press this button every second (or every millisecond, or whatever was necessary).

This hidden action idea is sufficient to treat timeouts in principle, but it is rather messy because it creates an enormous number of states to keep track of how many times the Tick action has occurred.

Another sensible approach, the one we'll use, is to imagine clever gnomes inside the device. These gnomes have a stopwatch, and decide, using their detailed and intimate knowledge of the device design, what actions the user is really doing. If the user presses the Off button, and the gnomes start counting. If the user holds the Off for three seconds, the gnomes say, "OK, that wasn't and off action, it was a really-off action!" If the user presses Off Off in quick succession, the gnomes say, "OK, that *was* a double-off action, not two separate off actions." Now the finite state machine (there aren't really any gnomes) has actions like one-off, really-off and double-off. Instead, the user's complicated, time-dependent actions are translated into the device's internal actions.

Henceforth, we will only talk about internal actions. We want to gloss over gnomes; we will talk about internal actions *as if* they are user actions. It is much easier, at least for most of our exposition in this book, to say things like the user presses a button Off, and the action off occurs, or presses a button Double-Off, and the action double-off occurs.

6.1.4 When is an action not an action?

Most actions change the state of the system in a straightforward way, but this is not always the case. A hotel keycard seems to unlock your room door, but it isn't as simple as that. When you first use your keycard after registering at the hotel desk, your card reprograms the room lock, so that the previous guest can no longer unlock the door (and of course your keycard won't reprogram any room other than the one you've been allocated). Now that the lock is reprogrammed, you can get in and out freely until the next guest registers and inserts their keycard on the lock. The previous guest can never reprogram the lock once you've started using the room; the lock only reprograms when a keycard is entered with a new code *and* the previous code. When your card is programmed by reception, they can program it with the previous guest's code; of course, the previous guest doesn't know your card's code, so they can't get back in or reprogram the lock. This cunning approach to hotel locks means that the reception desk doesn't have to keep reprogramming locks when somebody walks off with a keycard by mistake: in fact, you the guest do the reprogramming for them, with no conscious effort.

An abstract view of the lock is that it has two states, locked and unlocked, and inserting a keycard is a simple action that unlocks the door. This abstract view doesn't fully describe what goes on the first time a guest uses the lock—however, for designing the user interface it's probably sufficient only to model what the user *thinks* they are doing. Users don't think they are reprogramming the door, and there is no need for us to worry about that.

Incidental interaction is when an action, like this, occurs as a side-effect of the user interacting, and therefore with no additional effort from what the user was doing. Normally, incidental interaction achieves something the user wants—it does here, though most users won't have thought out the details of the room security that they need.

▷ See section 12.2 (p. 410) for further discussion of incidental interaction.

Nevertheless, knowing how the door lock works is crucial to how the hotel staff use the door. For example, if a porter carries your luggage to your room when you first register at the hotel, it's possible that the porter will use their masterkey to let you in. If so, the door lock will not have been reprogrammed to your card—there is a window of opportunity for the previous guest to return and steal your baggage. As usual, the simplest way around this flaw in the design is to train the user, in this case the porter, to ask you for your key to open the door. This not only tests your keycard works, but (if it does) it reprograms the lock for you.

6.2 Drawing state machines

Finite state machines have the terrific advantage of being very easy to draw and understand. They are therefore an excellent way to help support the interaction programming process: draw diagrams and think about them.

Actually, as we shall see, there is a caveat. Simple finite state machines are very easy to draw, but most finite state machines we want to use are not simple. They

are huge and practically impossible to draw (unless you use a computer), and certainly impossible to understand and follow. Fortunately this is a problem we can leave for later. For the time being, we want to draw and learn how to understand simple devices. Later we can worry about how to handle more complex devices, which we may not want to draw at all.

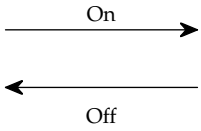
- ▷ Chapter 7, "Statecharts," describes a better way of drawing devices, that handles more complex designs; chapter 8, "Graphs," describes the generic theory that can handle anything; finally, several chapters—particularly chapter 9, "A framework for design"—show how to write programs that can explore and answer design questions.

6.2.1 A bright example

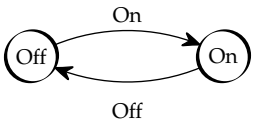
A flashlight is either off or on. We represent these two simple states as two circles:

To understand a finite state machine diagram, imagine a counter or bead. You are allowed exactly one counter, and it must be in the circle representing one state; the device is then in that state. When an action occurs, you move the state along the corresponding arrow reaching from the state it is in to the new state.

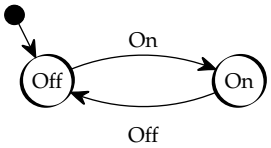
To move the counter, we need to introduce arrows. The actions the flashlight allows are coincidentally called on and off, and we represent these actions with two labeled arrows:



We can put the states and actions together to make a complete state machine diagram. We've bent the arrows to make a more pleasing diagram—the shape of the arrows doesn't matter, though.



The flashlight can be in two states, but which state is it in when you get it? In general, we need to indicate the starting, or default state a device is in when we get it. By convention, we indicate the default state with a special sort of blobbed arrow, or default arrow thus:



In other words, this flashlight will start its life in the off state. The meaning of this complete diagram can be spelled out:

- When we get the machine (the flashlight), it will be in the off state. We show the default state by the blobbed arrow.
- If the machine is in the off state (left hand circle), then doing On will take it to the on state.
- If the machine is in the on state (right hand state circle), then doing Off will take it to the off state.
- The diagram doesn't show what happens when we do On in the On state, or Off in the Off state. By convention, if there isn't an appropriate arrow for an action, the device does nothing—but this is a convention to stop diagrams from getting visually cluttered, not a conceptual convention.

Often in this book, to avoid visual clutter, we won't write down the labels on all arrows.

6.2.2 A darker example

The gun makes a very good example of an interactive device where clear, if not stark, usability and safety issues arise in simple machines.

In 1994, a seven year old, Brandon Maxfield, was shot in the face, leaving him a quadriplegic. He was shot accidentally with a Bryco Arms Model 38 semiautomatic pistol.

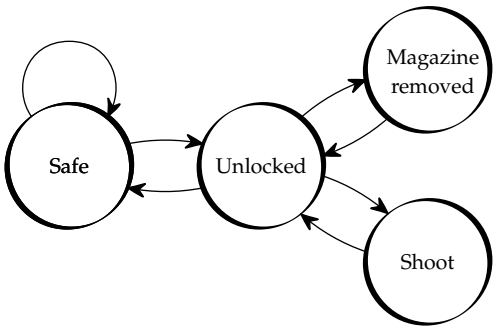
Pistols should have safety catches to stop them firing. When the safety catch is on, that is in the "safe" position, any action on the gun does nothing. The diagram below shows that if a gun is safe, *any* action keeps it safe.

From the diagram, it's clear that every action anybody can do in the safe state takes us along the arrow and back to the safe state—nothing ever happens. This isn't quite what we had in mind, since it isn't possible, as it's drawn, to get out of the safe state and do anything else. Even releasing the safety catch, as drawn, takes us back to the safe state.

Clearly the safe state needs an arrow coming out of it for when the user releases the safety catch, putting the gun in the fire mode. When the safety catch is off, the gun can shoot. The arrow coming out of it for the safety catch release action will go to an unlocked state, where the gun can do more things.

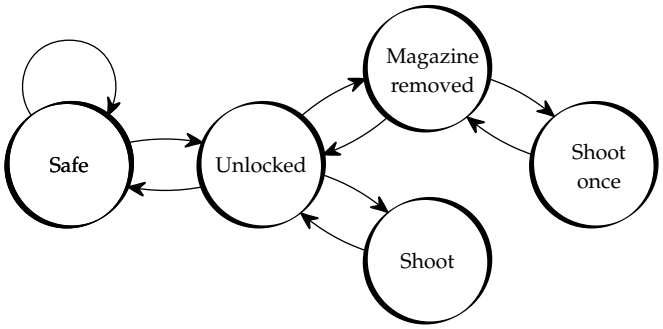
In the next, more accurate, diagram the safe state has *two* arrows leaving it: one for the safety release action, the other for every other action (which still do nothing).

In the case of this shooting, somebody was trying to remove the gun's ammunition clip, or magazine, so that the gun would have no bullets in it. Unfortunately, the design of the Bryco requires that the safety catch must be off in order to remove the magazine. So to empty the gun, you first have to release the safety catch; by design you *must* get the gun into the unlocked state to do this—and in the unlocked state you can, perhaps by accident, pull the trigger and go to the shoot state. From the shoot state, the gun fires and returns itself to the unlocked state when you release the trigger. So, the arrow from shoot to unlocked is done by the gun, rather than by the user, as all the other arrows are.



For most guns, there may still be a bullet left in the gun chamber when the magazine is removed. This is an intentional feature, so the gun can still shoot while the magazine is out and replaced or reloaded.

Here is a more accurate diagram, then, representing this possibility:



There's still an error in this diagram, as it incorrectly gives the impression that you can *continue* shooting with the magazine removed. The shoot once state returns to the magazine removed state, and the state diagram strictly does not say you can't go back to it and shoot again—it's only the name that says it can only shoot once in this state. In fact, even with the magazine in, you cannot shoot indefinitely—the gun runs out of bullets. The diagrams are not drawn with sufficient detail to count bullets; instead, they just show the "ability to shoot."

A basic rule of gun safety is that you should always assume a gun has a bullet in it, and that's the behavior the diagrams show too. In practice, you should also assume the safety is off too, but here the diagrams do not assume that, since they show the safe state.

The gun would have been safer if the magazine could have been removed with the safety catch engaged; arguably, it was a design flaw that the safety had to be released. The gun would have been even safer if the magazine could *only* have been removed with the safety catch engaged. However, for a gun that might be used in fighting, it is useful to be able to remove the magazine with the safety off, so you can shoot the remaining bullet in the chamber while you are reloading.

The designers had to decide whether the value of being able to shoot once in such circumstances outweighs the value of the gun never shooting accidentally when the magazine is removed; or they could have considered the cost and benefits of another safety mechanism (which might have given the user the choice between "cannot shoot with magazine removed" or perhaps "cannot shoot at all" and "can shoot with magazine removed").

▷ Making a device safer may make it more complicated, which in turn may make it harder to use and hence, ironically, less safe. Chapter 12, "Grand design," discusses how we can make more sophisticated devices without making the design more complex.

For guns, which are mechanical, every feature has a direct cost, both in manufacturing costs and in the risk that the mechanism may wear and become faulty. In contrast, adding features for electronic devices has essentially no cost except that the user interface becomes more complex—especially if a feature is added without adding further controls, because then existing buttons have to have more meanings to cope with the new feature. So, the design tradeoffs are very different, but the underlying theory is the same.

■ ■ ■

Some people argue that the best safety device for a gun is the user—that is, a sensible user would never aim a gun at somebody (unless they *wanted* to shoot them), so the child would not have been shot accidentally. This is the "guns don't kill, people do" argument. It is an argument that encourages gun manufacturers to blame the users and avoid improving their products; obviously to admit otherwise would be to expose themselves to all sorts of liabilities. If the same argument were applied to user interfaces more generally, then the user would be to blame for all problems, and this would similarly create a culture where nobody wanted to improve design for the sake of users. Of course, this is a politically charged argument; but so is usability.

▷ Car manufacturers took a similar position ("cars don't kill, drivers do"), until the exposé by Ralph Nader in the 1960s; see box 2.3, "Ralph Nader" (p. 51).

6.2.3 Basic rules for drawing state diagrams

Diagrams only make sense if they are drawn to adhere to certain conventions. State diagrams follow the following conventions—though as we shall soon see

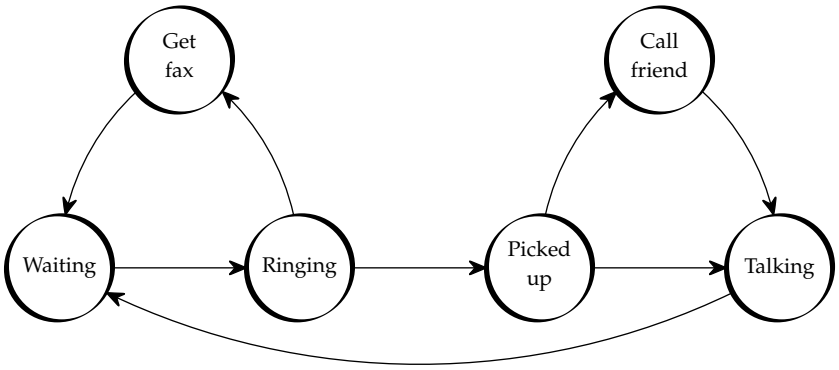


Figure 6.4: State diagrams do not have to describe devices and machines; they can describe what people do too. Here is a state diagram that says what you *and* a phone can do.

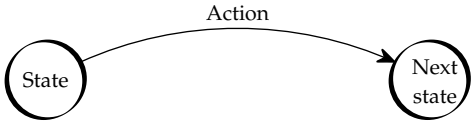
there are other ways of drawing state diagrams that will allow us to relax some of the rules when drawing. Behind our casual drawings there is always in principle a rigorous diagram drawn strictly according to these rules the system—if not the actual diagram—*must* obey the rules.

There are nearly twenty rules. The rules are all pretty obvious once you think about them. It is surprising how easy it is to build a device and accidentally forget some detail that these rules would have caught. The rules make sense for drawing *and* for using a device, but the rules are not visible in ordinary programs. It is possible to write a program that makes a device work, but for it to have elementary mistakes that go undetected.

1. Circles represent states. Sometimes rectangles are also used in this book.

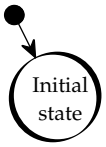
2. States are never drawn as overlapping or intersecting.
3. Arrows represent actions. Arrows do not need to be straight.

4. Although it is sometimes unavoidable, arrows shouldn't normally cross.
5. States and actions are labeled, but if we are worried about the visual clutter in the diagrams, we may omit some or all of the labels.



6. Every arrow starts and finishes at the circumference of a state circle.
7. *Every* state has exactly as many arrows pointing from it as there are possible actions, one arrow for each action. For example, if a device has b buttons, every state will have b arrows pointing away from it. Arrows coming out of a state have different labels.
8. Arrows can start and finish at the same state; these are called self-arrows. For clarity, self-arrows are often omitted from a diagram, because they are implied by the previous rule.

9. A state with all arrows from it returning to the same state are called terminal states; they are states where the system jams—which is undesirable for interactive systems. They are (almost always) an error, for they represent a state where no action can do anything and nothing further can be achieved. Explosive devices, fire extinguishers, and a few other single-use devices, typically which destroy themselves, have terminal states by design.
10. If we are trying to avoid visual clutter by not showing action labels, we may merge arrows going between the same states.
11. If two or more arrows go to the same state, it is occasionally convenient to join them together before they reach the state, to reduce the visual clutter. Otherwise, arrows are never coincident over any part of their length.
12. The initial or default starting state (if we want to indicate it as such) is pointed to with an arrow coming from a small black circle. Only one state is indicated as the initial state.



13. States with no arrows or default arrow going toward them represent states that can never be reached. They *always* represent an error (or possibly that the designer has not yet finished the device).
14. Finally, with the exception of single-use and similar devices (see rule 9 above), it must be possible to follow a sequence of arrows from any state to every other state. This important requirement is called strong connectivity. If a state machine is not properly connected in this way, either there are some states that cannot be reached by any sequence of actions, or there are some states (more generally, sets of states) that a user cannot escape from back to the rest of the device.

Box 6.3 Mealy and Moore machines There are many variations of finite state machine. The simplest kind merely recognize a sequence of actions, and are sometimes called finite state acceptors (FSAs). Finite state acceptors are familiar to programmers in the form of regular expressions, which are a concise and common way of specifying string patterns: regular expressions are compiled into finite state acceptors that then "accept" when the regular expression has been matched.

▷ Regular expressions are used in section 12.6.4 (p. 432).

Unless we are modeling a single-use device that does only one thing, like matching strings, these acceptors are not very useful for us. In this book, we are concerned with finite state *machines* (FSMs) that can provide output as a user interacts with them—an interactive device is only useful if it does something. For example a microwave oven must be able to cook, and a mobile phone must be able to allow the user to manage a call.

The two main sorts of FSMs are Mealy machines and Moore machines, named after their inventors. They differ in when they handle actions. In a Moore machine, the device does things *in* states, so a Moore machine may have a cook state or a call state. In contrast, a Mealy machine does things *between* states, on its transitions. When the user presses a button a Mealy machine does its action *as* it goes to the next state. The action of a Moore machine depends only on the current state, whereas the action of a Mealy machine depends on the current state *and* the user's action, because the effects happen on the transition.

Although the two approaches are equivalent, a Moore machine requires more states (that's how to remember which is which) because a Mealy machine can use different transitions—even to and from the same states—to do different things, whereas a Moore machine must have different states to do different things. This book uses Moore machines.

If we allow the number of states to be unlimited, then we have a labeled transition system (LTS). Of course, FSMs are finite cases of LTS, and some people use the terms almost interchangeably. Another variation is to make FSMs nondeterministic, so that they can effectively be in more than one state at a time; this makes them more concise, and you would think, more powerful. However, nondeterministic FSMs (NDFSMs) are mathematically equivalent to ordinary deterministic FSMs, because the states of an deterministic FSM can represent the possible sets of states of a nondeterministic FSM.

- ▷ Strong connectivity will be defined properly in chapter 8, "Graphs." Strong connectivity is not easily checked visually but we'll see how to check for it using a program automatically in chapter 9, "A framework for design."

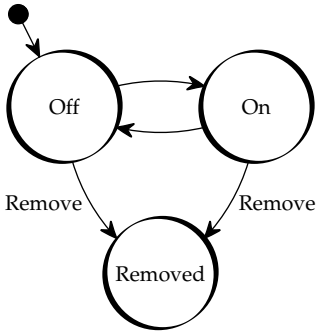
Some of these rules are aesthetic, but the most interesting rules are, like the last, ones that we can check by program.

6.3 From drawing rules to interaction laws

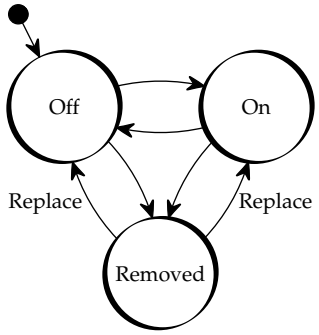
Our rules seem to be about *drawing* diagrams, but the rules also have an effect on the *behavior* of interactive devices. For example, if several arrows could come out of a state all with the same label (which we don't allow when we are drawing a diagram), this would mean that when the user does the action associated with

that label anything could happen—we don't know which arrow the device would follow. This would mean that the device is nondeterministic: a user could not predict what would happen with a certain action. It's easy enough to detect when this problem arises by writing a program that examines the arrows (as we'll see in later chapters), but designers have to decide what to do. Is it merely a design error or oversight? More likely, *we* have not decided what to do, and the nondeterminism indicates that we haven't thought enough about the design.

The next diagram shows a flashlight with a removable lightbulb.



We can now remove the lightbulb, but as drawn above, we would get stuck with the lightbulb removed and be unable to do anything else: there is no way out of the removed state as the diagram is drawn. Let's add replace arrows, as in the diagram below. (I've removed the other arrow labels, as they'd make the diagram too cluttered.)



We're breaking our rules: we need to say which state the flashlight will be in when the bulb was removed when it was off, and one to say which state it will be in when the bulb was removed when it was on. In short, when we replace the bulb, the flashlight might be on or off. We have the problem! When the user does the action "replace," the bulb might end up being be on or off, *and we do not know which*, that is, this device design is nondeterministic.

Once the problem is detected, we then have to decide what to do. Perhaps we could make the state machine more realistic: actually, the bulb can be removed when it is on or off, so the single state is actually two states—this might be fine for a low-voltage device. Or perhaps we should modify the design, so that when

a bulb is removed the state of the device changes to off, even if it was on. This is a nice solution: it removes the nondeterminism and makes the device safer as well, particularly if the device uses high voltages (like an ordinary domestic light, which of course has the same transition diagram). Or we could make the device have "no user serviceable parts" and simply make it impossible to remove or replace the bulb—this might be a good solution if the light bulb was a LED, which have very long lifetimes and are usually fixed in place, since having no socket makes the device even cheaper.

This is an example of a drawing law is a law about interaction programming. Indeed, once we start thinking of laws for drawing or designing interactive devices, it becomes obvious that there are many more that can be checked.

6.3.1 Avoid the reset escape

On some badly designed systems, the only arrow coming out of some states is called reset. A big button called RESET would be far too easy to press by accident, so the reset button is usually a little hole around the back of the gadget, which may well require a paper clip or pin poked in it to operate it.

I think the reset button is an admission of failure by the designer: "this machine may jam and we want the user to be able to reset it." The reset button makes it easier to make shoddy products—why bother to check the system design thoroughly if the user can *always* press reset and recover from anything that has gone wrong? On the other hand, a reset button is a better option than occasionally having to remove the batteries to do the reset.

6.3.2 Have consistent action names

Suppose we have an action called Off. Clearly, this should be represented as an arrow that points to (possibly more than one) state called off. If we checked that, we would know that the thing we were building had a reliable Off button.

6.3.3 Avoid pointless states

If a state has only one arrow coming out of it (all the other arrows looping back to the same state), the user has only one possible action. If there is no change in indicators, then the user sees no change out of being in this state. In which case, what is the point of the state? The next thing must eventually happen, and all the user can do is say when. The only way they can stop the next state happening is by walking away—and then someone else might do it.

Almost certainly a state with a single arrow coming out of it is a design mistake: it gives the user no choice (other than to delay). If we wanted the user to confirm a choice, then there would need to be at least two arrows: one for yes and one for no (perhaps more). Certainly any state with a single arrow coming out of it can be

found automatically in the design process, and questions can be asked about what it is doing: if there is no time delay issue, then the state should be removed.

Note that during use, arrows can disappear if states or actions become impossible of forbidden. In other words, a check like whether the number of arrows leaving a state is one (or less!) is a check that may need to be done dynamically, while the device is in use.

All laws have exceptions. Although a state may allow the user only one choice—apparently no choice at all—and therefore be unnecessary from a choice point of view, there might be an opportunity to tell the user something, to help them understand what is about to happen, or there might be legal warnings.

An email program, to take one example, might want to pop up a dialog box that tells the user that new mail has arrived; this might be a case where the only response is OK—but why not OK *and* read it? It's a design choice that we could easily uncover automatically by checking the state machine design.

If there is only one exit from the help or legal state, the user cannot then change their mind. So in what sense is the help helping? Surely, if a user knows or understands something better, they might realize their choice was a mistake. So the state should not be removed so much as provided with an alternative exit: say, cancel.

6.3.4 Don't be rude

An obvious rule of good human behavior is this: don't be rude. Imagine asking the waiter at a restaurant if they have coffee. "Yes, what would you like?" You have a discussion about what they can do: latte, capuccino, americano ... and you ask for a filter coffee. "With milk?" Yes, you'd like to have milk, you say. Then the waiter says, "We don't have any coffee!"

This human dialog is rude and disappointing: the waiter must have known all along that they couldn't serve you coffee, however you liked it. But they kept on asking questions until you had made your mind up, and then they told you it was all impossible! Whereas this behavior would be exceptional between two humans, regretfully it's routine in interactive devices. Very often the device provides options that the designers hoped or expected would be available, but for some reason do not work fully when the device is being used.

Devices should always check that the "destination" state can always be reached with the choices being offered to the user—this is a dynamic check, since some resources (such as coffee) can disappear while a device is in use.

The Sony Ericsson T610 mobile phone, for example, tells you when you have missed an incoming call. It lists all your missed calls—there may be more than one—and has two responses indicated: Call and More. Unfortunately, More may not do anything. Many systems provide menus of features, with some features available on the device, and some on the internet. Mobile phones (like the Sony Ericsson) have a menu hierarchy that is partly on the phone, partly on WAP ("wireless application protocol"—a sort-of simplified web service). Users cannot tell which is which until they select an offered option and the phone then tries to connect. The phone may be useless while it is connecting. The phone might not have

WAP enabled (because the user has decided not to pay for WAP services), so it is never possible to get any WAP feature. Yet the phone will still try to connect.

There are two solutions. The simplest solution to all of these problems is for the device to check that a user can complete all parts of a transaction before allowing it to start. Perhaps the unavailable options should be grayed out; trying to select them would bring up an explanation. Perhaps they would not be shown at all (depending on whether this is a permanent or temporary state of affairs). Alternatively, users could be warned that they won't be able to complete the activity but the device can "take notes" so that when the service becomes available it can be done automatically from the notes. This is like a more helpful waiter saying, "Thanks for your order. We're out of coffee right now, but I will go and get some from a shop myself for you and be back as soon as possible." Polite restaurants like that become famous.

The "don't be rude" rule is really a generalization of the drawing rule that every arrow must start and finish at a state circle. A device should never trap the user in a sequence of arrows they cannot get to the end of. Indeed, many useful interaction programming laws come from checking routes through the system design, rather than just checking states and arrows alone.

The "don't be rude" (or "do be polite") rule is bigger than just button-operated devices running simple state machines. Figure 6.5 (p. 187) shows how well-designed menus in computer desktop applications are polite by using visual and typographic conventions: they show what is possible before the user even starts to try. Whether a menu item is active or grayed-out is worked out dynamically, depending on the current state and what is reachable from it, just as it should be in pushbutton-controlled devices like mobile phones.

▷ Section 11.3.1 (p. 383) introduces some polite pushbutton-type device techniques.

6.3.5 Don't be misleading

Being rude is a property of a device that is being rude. The device, or the designer analyzing the device, ought to know whether it is rude or not. On the other hand, whether a device is misleading or not, another unwelcome feature, really depends on where the user feels led in the first place. It's an example of an important usability property that isn't just a property of the device itself, but depends on the user too. Here's a story that makes the point.

I go to a public washstand to wash my hands. I put soap on my hands, and try to turn the water on. Of course, now I have all this soap on my hands, I *have* to wash them. Although there is a knob, it doesn't turn on the water; no water comes out whatever I do. In fact, the whole thing is loose—indicating that people before me have forced it one way or the other. I try the next knob, as there are five of them in a row. Same problem on all of them. Somebody else now tries to get one to work too, and they are have the same problem as I do. So, I'm not stupid, at least. Perhaps there is no water at all?

Figure 6.5: A desktop application menu (taken from running on Apple's OSX; in fact, Grab is the program that makes screen shots, like this one). Choosing a menu item is like pressing a button: it will usually make the application do something. Notice how good practice is to use visual conventions so that the user can see easily whether: (*i*) a menu item will work immediately, such as "About Grab" and "Hide Grab"—the latter has a keyboard shortcut as well; (*ii*) a menu item will bring up further menus to select actions from, as in "Services," because of the triangle symbol; (*iii*) the menu will bring up a dialog before it does something, as in "Preferences..."; and (*iv*) the menu item currently does nothing, as in "Show All," which is grayed out.

And then we notice a paper notice; hand-washing is automatic. If you put your hand under the outlet in *just the right place* the water flows automatically. The knob is not one of the places that makes the taps come on. Now there's water, we soon find out that the knob controls the temperature of the water, not whether it is flowing or not.

Enough people have been misled by this design that there needs to be a notice, and indeed there is a sign above each knob. But who thinks of reading a sign, that isn't even where you are looking, to tell them how to use an everyday object?

The misleading here is that the knob allows the user to do an action, and a conventional action that would normally get water. It has the affordance of getting water, in fact. But we are wrong, misled. Once misled, we form a different model of what is going on: the device is broken and/or there can't be any water.

▷ There's more on affordance in section 12.3 (p. 415).

It's clearly a bad design, because the owner of the place found it necessary to put up a sign to help users who were misled. We do want to design interactive devices that mislead; at best, it wastes a lot of time. But whether a user is misled depends a lot on the user's expectations, prior knowledge and so on. Generally, you have to do experiments with real users to find out what they expect before you can be sure a design is alright.

6.3.6 Don't be secretive

Another rule of good behavior is not to be secretive; being secretive frustrates people. Similarly, a device can be secretive and frustrate its user, particularly if there is only one way to do something, and that way is not very obvious.

The Alba TVD3406 is a television with a remote control with the usual range of buttons and an on-screen menu. Neither the remote control nor the on-screen menu seems to provide any way to tune in the TV. In fact, you need to press TV Menu *twice in quick succession* to be able to start tuning the TV. The on-screen menu provides help, but it doesn't say, "If you can't see what you want, try pressing the button twice quickly." In short, you can't tune in the TV without reading the user manual—despite having an otherwise nice on-screen menu system; the TVD3406 is unnecessarily secretive.

6.3.7 Check designs automatically

It is unreasonable to expect users to sort out design problems themselves—forcing the user to recover from the errors themselves, that is, if they notice them. Even in experimental situations where they are paid to help the designers, users may find it hard to describe what problems; far better to use an automatic tool and eliminate as many errors as possible before humans start working on it. Fortunately, checking almost all laws can be automated.

Once automatic design tools have been built, they can be used again and again with no further effort. If we rely on human user testing, every time the design is modified, we'd need to spend another few hours testing with people to see whether we've still got a decent design. It is far better to eliminate obvious errors automatically and speedily, and to save the users for more insightful things.

Imagine building a system with b buttons: it would be a good idea to check automatically, of every state, that there are b arrows coming out of it. If not, some buttons won't work in some states. This is next to impossible for a human to do, but it is trivial to do automatically, and of course, checking automatically (and hence rigorously) increases the quality of any device we build.

If we have such a design tool, when we think of a new design property, whether important or just interesting, we can add checking it to the design tool's repertoire. Immediately, this check becomes available for every design the tool is applied to.

▷ Section 10.7.4 (p. 356) suggests checking for error messages in source code. From chapter 9, "A framework for design," on we'll introduce more ideas for design tools.

6.3.8 Provide uniform help

If a device provides help, is help available in *every* state? We could imagine an automatic tool that reports the states where help is (or isn't) available. If the device says there is no help when it is off, perhaps the designer would tick a check box on the design tool, as it were, to say *this* problem is OK; this state is an exception where the rule does not need to be enforced. If we check this law, we then won't need to draw the "parallel universe" of the help states, because we know that every state has its own parallel with help.

6.3.9 Dynamic laws

There is not just an obligation for you, as a designer, to check your new designs *before* they are used—but also for the devices to recheck themselves *while* they are in use. Some features or services may unpredictably become unavailable (say, if the network goes down, or the user's credit runs out, or just when the CD music track finishes); if so, the user interface needs to change dynamically; it must not rudely provide the user with apparent options that will fail. Indeed, the restaurant that suddenly runs out of coffee has got to change how it interacts with customers; otherwise it will unexpectedly become rude.

6.3.10 Make generic devices

A device might be designed to be used in two or more countries that use different languages. Are these devices exactly the same, despite the different names and texts associated with all their states and actions?

Device equivalence can be achieved in two different ways.

- Build two or more devices, then compare them.
- Build a generic device, and then make different versions of it.

It is perhaps slightly harder to make a generic device, but once it is done, it can be extended to new languages or applications very easily. An automatic tool should be used to check that the generic bit of the device applies in all states, just as we might check help is available in all states.

6.3.11 Define your own laws

If you are designing a device, what laws should it obey? What laws would help a user? What laws would avoid bad outcomes—irritated users, dead users, lost work? As the examples of rudeness and secrecy show, it is not hard to think of sensible laws of interaction that make a design better and that are fairly easy to understand in terms of interaction programming.

It's not difficult to think of your own laws. Here are some more examples, which need working out a bit more, depending on what your users and device are trying

to do. Don't make things too hard—which begs the question of how hard things are to do with the device being designed. Provide undo, so users can recover from mistakes—it's surprising how many devices have no undo at all. Make it compatible with a previous device—so if a user doesn't know about new features, it doesn't matter.

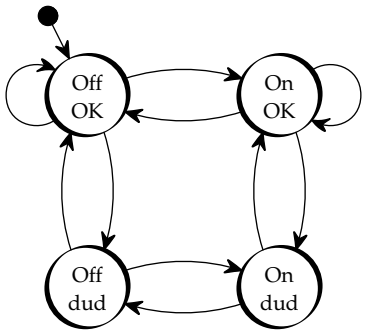
▷ We'll give lots of examples of measuring how hard things are to do in later chapters, particularly chapter 9, "A framework for design." Chapter 5, "Communication," gave examples of automatic ways of making design tradeoffs over how hard things are to do.

6.4 A four-state worked example

A flashlight (torch) can be off in lots of different ways. Let's consider a dud bulb and an OK bulb. There are many ways of drawing the device, but first we use a table to define all the states and actions—bearing in mind that some things are not possible, for example, you can't switch it on if it is *already* on, though you can replace a bulb even if the bulb is OK.

		—*Actions*—			
		Switch on	**Switch off**	**Bulb dies**	**Replace bulb**
States	on & OK	⟨*not possible*⟩	off & OK	on & dud	on & OK
	on & dud	⟨*not possible*⟩	off & dud	⟨*not possible*⟩	on & OK
	off & OK	on & OK	⟨*not possible*⟩	off & dud	off & OK
	off & dud	on & dud	⟨*not possible*⟩	⟨*not possible*⟩	off & OK

Here it is, drawn as a diagram. In all we need four states, and lots of arrows:



When I first looked at this diagram I wondered why it wasn't symmetrical. In fact, it could be: there is no reason in principle why a user can't replace a dud bulb with a dud bulb! If this action is allowed, then the diagram would have four loops rather than two.

A flashlight is very familiar and pretty trivial as things go. There aren't many ways to design them badly given that most of what they have to do is dictated by just linking up the basic states they require to work as a complete device.

Box 6.4 Bad user interfaces earn money When you leave a house with a burglar alarm, you enter your secret code and the alarm gives you a minute or so to get out of the building before the alarm is activated. The delay is a time out and can cause problems.

Georgia Institute of Technology has an "aware house" (www.cc.gatech.edu/fce/ahri) full of exciting gadgets—and it has a conventional burglar alarm, made by Brinks Home Security, the sort that could be used in any domestic house. What's unusual about this system is that lots of people use the house.

Here's the problem. When you arm the burglar alarm just before leaving the house, the alarm warbles for ten seconds to tell you what you have done, then it gives you a minute's grace—the timeout—to get out before the alarm is armed. So you leave. But this is a busy house, and someone else enters within that minute's grace, without the alarm making any noise—just as if the alarm wasn't set. Maybe they settle down to read a magazine in the living room ... Meanwhile, the alarm gets itself fully activated. It makes no noise that it has changed state.

Time for a cup of coffee? Oops, when they get up to walk to the kitchen the alarm goes off. They have to rush to the control box to key in the secret code. Hopefully they can remember the code in their panic! If not, the security people who rush to the scene to "catch the burglar" charge for their services.

Thus the company that supplies or supports the burglar alarm makes a profit out the bad user interface design. There is no incentive to improve a design when you can blame the users for the problems *and* make money out of it!

▷ See box 6.2, "Timeouts" (p. 174) and the discussion at the beginning of this chapter for more on how to avoid timeouts.

6.5 A larger worked example: an alarm clock

Let's try something more complex than a flashlight. A digital clock has states for each time it can display (60 minutes times 24 hours means 1,440 states). The number would be 60 times bigger if we (or the alarm clock) worried about keeping track of seconds.

The basic action of the clock is to tick. Each tick takes the clock from one state to another, and the time shown on the display shows which state it is in. There is nothing wrong with a device doing actions to itself.

We could imagine a gnome (small enough to fit inside the device) doing the tick action if we really want "users" to do all the actions and for the gadget to be a "pure" state machine. The gnome would have to do the tick action reliably every minute, day and night, and it would be such a boring job that it would soon be automated. The gnome could be replaced by investing in a metrognome as a matter of sound ecognomics (at least if you pay gnomes an hourly rate) as well as of sound ergognomics—as we wouldn't want our gnome getting repetitive strain injuries from all its button pressing.

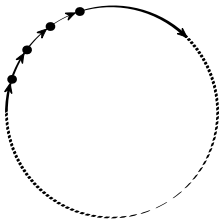
An alarm clock, further, has to "know": whether the alarm has been set and what time the alarm should go off.

In principle, an alarm clock does not need to know whether the alarm is ringing, since this can be worked out from the time it is showing, the time the alarm is set

for (if it is set), and how long the alarm is supposed to ring for. Ringing is an effect, not a state.

Next, the alarm clock could have a button so the user can switch the alarm off earlier. Hence there is another factor to add to the list above: whether the key Snooze has been pressed.

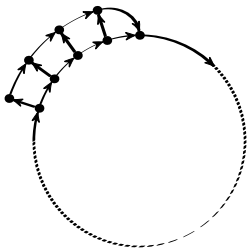
We could carry on adding features to our clock; instead, we'll start to analyze what we have designed so far. Rather than analyze the full device, let's first imagine a "broken" alarm clock with a *fixed* alarm time, say 08:30. (Eventually we will need lots of these clocks, all "broken" at different times, so we can eventually change the alarm time by choosing the right clock!) This alarm clock must be able to display all $1,440$ times, from 00:00 to 23:59, so it needs $1,440$ states:



Here we've shown a state diagram but only drawn four of the $1,440$ states explicitly; the dashed line is supposed to indicate all the missing detail.

Each tick of the clock takes the clock to the next state, going around the circle, and the display changes the time shown, say from 12:39 to 12:40, then to 12:41, 12:42, and so on.

When this alarm clock gets to the state corresponding to 08:30, the alarm is supposed to ring for that and the next three ticks: so there are four states (08:30 to 08:33) where the alarm rings. However, in each of these four states, the user might press the Snooze button and silence the alarm. So we need to add some more states, getting $1,440 + 4 = 1,444$ states so far.

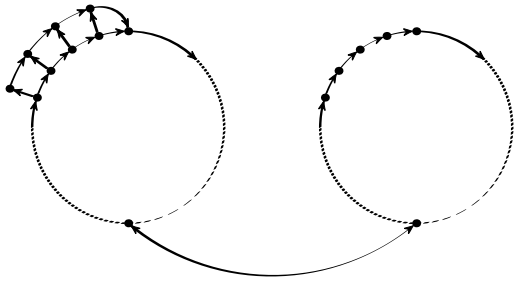


In the diagram, we've drawn the four snooze states outside the main circle. The idea is that the gnome's ticks on their own take the alarm clock around the inner circle. At 08:30 we're at the first state explicitly shown on the inner circle, and the alarm starts ringing. If the user does nothing, the next action will be a tick, and we go one state further around the inner circle—and the alarm continues to ring. If the user presses Snooze, we would follow an arrow to the outer arc of states. In these states, the alarm does not ring, but as each tick occurs we stay moving along the outer arc, and thus avoid going to any state where the alarm rings. Eventually, after 4 ticks in the snoozing states, the clock goes back into a state on the main circle. Until we get back to 08:30 again the alarm will be silent.

Let's add another feature and so we can lock the alarm off, so it never makes a noise. Thus the 08:30 alarm clock needs another 1,440 states to show all the times when the alarm will never ring because it is off in addition to the original 1,440 states when it can ring.

Why can't it have on/off with just one or two more states? If the alarm clock had a single state for when the alarm is off, then it could not keep track of the time when the alarm was off. Remember, by our rules, we could only have one arrow coming out of this single state labeled ON, and it could only go to one time with the alarm on. So we need 1,440 states, all for alarm off in each of the 1,440 different times.

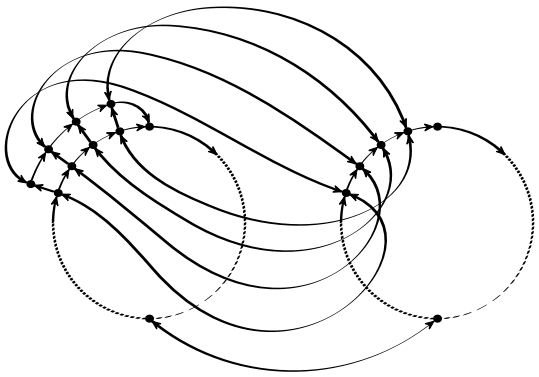
These 1,440 states are shown on the right of the diagram below. (As before, I have only shown a few of the 1,440 states explicitly.)



When the alarm is switched on, the clock goes from showing the time (perhaps 11:40) to showing the same time and the alarm being set. The double-headed arrow at the bottom shows what the alarm on/off button does.

If we have been pedantic, there would have to be 1,440 such arrows, to allow the alarm to be switched on and off in each of the 1,440 pairs of states. Really, there are 2,884 arrows, since there is one facing each way plus 4 one-way arrows from the snooze states—these 4 arrows are one way, because when the alarm is off it can't get back into snoozing, so the righthand circle has 4 fewer states than the lefthand circle.

Below, we've added in *just* the arrows to and from the alarm ringing and snoozing states to the alarm off states:



This last diagram shows the following:

- If the alarm is ringing, we can press Snooze to switch it off. Alternatively, we can switch the alarm off and we go to the corresponding state on the right hand side circle of states, where the alarm does not ring. And ...
- If the alarm is not ringing, there are two possibilities:

  (*i*) If the alarm *is* set (we are somewhere on the left, except for the four inner states where the alarm rings), switching the alarm off takes us to the corresponding state on the righthand side.

  (*ii*) If the alarm *is not* set (we are somewhere on the right), switching the alarm on goes from the righthand side to the corresponding state on the lefthand side. But note that if the time is currently 08:30, 08:31, 08:32 or 08:33, then putting the alarm on takes us to the inner circle of states, where the alarm rings.
- The diagram omits showing 1,435 arrows in each direction. It also does not show 2,870 states.

So an 08:30 alarm clock, with the possibility of switching the alarm off, and of having a snooze button that immediately silences the alarm requires 2,884 states. We've only sketched what it would look like.

In reality we need an alarm clock that lets us change the alarm time. In fact, we need an 00:00 alarm clock, an 00:01 alarm clock, and so on. That comes to 1,440 alarm clocks, all much like the 08:30 clock but different.

There will be lots more arrows. Imagine we are running the 08:30 alarm clock and we want to increase the alarm time to 08:31. That means we must have arrows for the "increase alarm time by one minute" action from each of the 2,884 states in the 08:30 alarm clock to corresponding states in the 08:31 alarm clock. In fact we need 2,884 arrows to do this.

It is likely that a real alarm clock would have more buttons to adjust the alarm time, for example, in steps of 10 minutes and 1 hour, as well as just 1 minute. The extra buttons do not require any more states, but each button requires many more arrows of its own—each new button we add to the clock's design in principle requires one arrow from *every* state since the user can press the button whatever state the alarm clock is in. That's one of our original rules for drawing diagrams.

Finally, putting all these alarm clocks together into one box makes a giant machine with $1,440 \times 2,884 = 4,152,960$ states. This is a huge number, and clearly if we designed an alarm clock by *drawing* the states and actions, we'd get into a mess.

6.6 Doing your own drawings

Understanding state diagrams really requires playing with them: try doing some of your own drawings. Find one of your own flashlights, and work out its diagram. You could add an action to account for bulbs blowing (burning out)—then

you would also need actions to replace dud bulbs with new, working bulbs. Draw in the extra arrows needed. How about doing a drawing for your car headlights, including full beam and dipped states? How do your fog lights work? (In Europe, you can have front and rear fog lights, separately controlled, but they can only be switched on when the main lights are on.)

6.7 The public image problem

Although we can't draw very big state machines, all the ideas are relevant to designing and understanding large systems. Indeed, because of the widespread confusion between drawings of state machines and machines themselves, the "public relations" problems with state machines have become dire.

- Any state machine you can draw is going to be small-scale and simple. So people mistakenly think state machines *are* simple.
- Almost any state machine that is interesting is too complex to draw. So people rarely think of interesting designs as state machines.

Regardless of their poor reputation, by drawing state machines, we'll get to understand them well, and we've seen how many problems about interaction programming can be understood and handled by treating devices as finite state machines. There are many good design ideas that become obvious when we exploit the clarity of state machines.

In short, if you don't understand state machines, you don't understand interaction. Once you understand state machines, you are welcome to program more interesting user interfaces: but don't lose track of the issues that matter.

6.8 The key design problem

If a user does not know what state a device is in, the device will be unpredictable; if the user doesn't know what mode a device is in, every action will have unpredictable effects. Unfortunately there are usually far too many states and modes to make it practical to tell the user which state or mode a device is in, or for the user to understand the difference between this mode or state and some fifteen others.

Much of user interface design boils down to balancing the device's need for many states and modes so that it can do a wide range of useful things for the user against the opposite constraint of making the modes clearly known or understood to the user, who should be in control—and with the constraint of the designer knowing what is going on too.

The key problem in interaction programming is that neither users nor designers can see states, modes, or actions (except in special, very simple cases).

With electronic gadgets, there are no visible features necessarily to be seen connecting actions to effects. In fact, inside gadgets there are, as it were, numerous features haphazardly crisscrossing. Because you can't *see* the mess of states, modes,

and actions inside, nobody bothers to make them straightforward. Adding a random action seems as good as any other action. With a large device, a missing action or an action doing the wrong thing is not noticed—until somebody needs it.

Although a designer might satisfactorily conceptualize a simple system reliably, devices can get big and there are no *general* answers to the problem. People are still out researching how to represent what's "important" in design and use.

A confounding problem is that most devices are programmed in ways that obscure things even further. In a typical programming language—like Java, C, C++, or even Prolog—what the program says as a piece of text and what that program does, how it interacts with the user, are very different things.

You cannot read a program and *see* how it will interact; instead, you have to work out what it will do when it is interacting with a user. You have to simulate how it will work, going through it step by step.

In fact, interaction is a side effect of all conventional programming. Just as global variables and goto statements are generally frowned on because what they do isn't clear, interaction is never clear from a program.

Functional programming, which tries very hard to be clear, *avoids* doing anything with side-effects, such as interaction, on principle because it makes programs too messy to understand! Being event-driven or object-oriented does not change the basic problem.

When a programmer changes a program, the effect on the user interface can be arbitrary. In particular, improving a program (making it shorter or neater, say) has an unrelated effect on its user interface. Conversely, improving a user interface can have an arbitrary effect on the program code; worse, what might seem like a little improvement to the user interface can turn into an insurmountable modification of the program—so no programmer will want to do it.

Designers and users cannot see what they are interacting with: they just see the changes. There is no "big picture." Nor can programmers see what users are interacting with, even looking at the program code—and they are in a slightly worse position because they may *think* they can see everything as plain as the program code before them. But the code is not the interaction programming.

If we insist on a simple representation for interactive systems, namely, one based explicitly on states and actions, we can pretty much eliminate the problems.

Because the theory of finite state machines is simple, we can build all sorts of things *reliably* on top of them—and we can confirm, in many ways, that we are designing what we intend to design.

Interactive devices themselves are of course one of the main things we can build, but we can also make user manuals and many other things, such as transition diagrams, that will either help users or designers better understand and gain insights into the design of the device.

The more views of a device we have, the more able we are to get a good grip on its workings—and the more people who can be involved productively in the design process. Phrased more carefully, the more *reliable* views of a device we have, the more able we are to understand it—since devices are complex we *must* generate views and alternative representations automatically. As we will see, we

can automatically generate working prototypes cheaply, so we can get users involved immediately, and we can generate draft user manuals quickly so we can get technical authors involved immediately.

In contrast, in a conventional approach to programming interactive devices, all of these things are very hard. To work out any sort of user manual requires a lot of careful—and error prone—thinking. Often, writing a user manual is therefore left until the program is "finished," since trying to write it any earlier only means that it will need expensive revisions as well. Ideas a technical author gets while reading or writing the manual—especially ways to simplify the design—are almost impossible to implement at this late stage.

- ▷ Well-written programs often simulate other styles of programming by building virtual machines. Chapter 9, "A framework for design," shows how conventional programming can simulate clearer models of interactive programs. Clearer models can then be used to create multiple views of interactive systems, for interaction, analysis, manual writing, and so on.

With a little care, then, we can arrange for insights from users or technical authors or other people to feed back into design. Since the state approach is simple to program, changes that are needed, say, in the user manual are *easy* to feed back into the core design. Thus we can have a *single* design, namely, a finite state machine, and all aspects of design work can follow from it.

There is only one thing to keep up to date, only one core thing to modify, rather than many independent versions—prototypes, scenarios, manuals, actual devices, and so on. We therefore end up with a more consistent, higher-quality design. This is concurrent design: when properly organized, many parts of the design process can proceed concurrently.

In contrast, in a normal sequential design process, you daren't make contributions to the design at the wrong time because your effort may be wasted; it may be calling for a revision to some earlier part of the process. In a sequential approach, there is no point working on the user manual too soon, because the device design might change. So, write the user manual *after* the device design is finalized. But then, if you have (or anyone else has) any design insights from reading the draft manual, you *cannot* afford to go back and revise the device design to help fix the manual.

Typically, a sequential design process has to keep users out of all technical parts of the design until practically the last moment, and then so much work has been done on the design that nobody is prepared to make many changes. Concurrent design is better, but it requires approaching design from a whole different set of assumptions.

6.9 Conclusions

What have we learned? We've defined states, modes, indicators, and actions, shown how to draw systems clearly, and how to get things done. Implicitly we have learned something very important: the user of a device, even as simple as an

alarm clock, hasn't a hope of being able to check it out—these things are surprisingly big when all interaction details are considered. When people buy a clock, they will have to take it on faith that it works sensibly—there is no way a user can explore and understand the thousands of state transitions. Even a paid user that we might employ as a tester of alarm clock designs *cannot* work hard or long enough to check a device reasonably well. Most gadgets are more complex than alarm clocks, so the user's problems are usually worse. Most device designs are not thoroughly checked.

If users can't understand devices in principle, then designers are obliged to be all the more careful and honest in the design process. The only way to do that is to use automatic design tools.

The cost of all these advantages is that state machines are simple, and some interactive devices would be very hard to represent as state machines. The counterargument to that is that anything that is too hard to represent in this way is almost certainly too hard to use (it would certainly be too hard to write about in this book)—that's because we've made a very good stab at solving the key design problem.

Finally, if we draw state machines, we won't want to draw all of the states and actions explicitly, and we could be tempted to take shortcuts (as we did throughout this chapter) to make the diagrams possible to draw without being cluttered with massive detail. We therefore need tools to support the design process; we aren't going to do it well enough by hand.

6.9.1 Further reading

- Feynman, R. P., *Feynman Lectures on Computation*, Addison-Wesley, 1996. This book has an excellent introduction to finite state machines. The book is well worth reading as a clear, general introduction to computer science too—even if you are an expert computer scientist, Feynman's exposition is very lucid and insightful.

One of the nice things about state machines is that the ideas were introduced so long ago that the research papers about them are now easy to read. These are some classics I recommend:

- Parnas, D. L., "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," in *Proceedings of the 24th. ACM National Conference*, pp379–385, 1964. Dave Parnas, now famous for his software engineering, introduced state machines for designing interactive computer systems.

- Newman, W. M., "A System for Interactive Graphical Programming," in *Proceedings of the 1968 Spring Joint Computer Conference*, American Federation of Information Processing Societies, pp47–54, 1969. William Newman went on to coauthor one of the classic graphics textbooks, commonly called "Newman and Sproull": Newman, W. M., and Sproull, R. F., *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, 1979.

- Wasserman, A. I., "Extending State Transition Diagrams for the Specification of Human Computer Interaction," *IEEE Transactions on Software Engineering*, **SE-11**(8), pp699–713, 1985. Tony Wasserman extended state transition diagrams to make them more useful for specifying interactive systems; he built many UIMS—user interface management systems, which took state machines as their working specifications. (There are lots of other extensions to finite state machines.)

    ▷ We'll develop a framework—our name for a "skeleton" UIMS—in chapter 9, "A framework for design," onward.